Assembler Syntax 2

The SunOS 5.*x* SPARC assembler takes assembly language programs, as specified in this document, and produces relocatable object files for processing by the SunOS 5.*x* SPARC link editor. The assembly language described in this document corresponds to the SPARC instruction set defined in the *SPARC Architecture Manual* (Version 8 and Version 9) and is intended for use on machines that use the SPARC architecture.

This chapter is organized into the following sections:

Syntax Notation	*page 3*
Assembler File Syntax	*page 4*
Lexical Features	*page 4*
Assembler Error Messages	*page 10*

Syntax Notation

In the descriptions of assembly language syntax in this chapter:

- Brackets ([ ]) enclose optional items.
- Asterisks (*) indicate items to be repeated zero or more times.
- Braces ({ }) enclose alternate item choices, which are separated from each other by vertical bars (|).
- Wherever blanks are allowed, arbitrary numbers of blanks and horizontal tabs may be used. Newline characters are not allowed in place of blanks.

2

Assembler File Syntax

The syntax of assembly language *files* is:

```
[line]*
```

Lines Syntax

The syntax of assembly language *lines* is:

```
[statement [ ; statement]*] [!comment]
```

Statement Syntax

The syntax of an assembly language *statement* is:

```
[label:] [instruction]
```

where:

`label`
is a symbol name.

`instruction`
is an encoded pseudo-op, synthetic instruction, or instruction.

Lexical Features

This section describes the lexical features of the assembler syntax.

Case Distinction

Uppercase and lowercase letters are distinct everywhere *except* in the names of special symbols. Special symbol names have no case distinction.

Comments

A comment is preceded by an exclamation mark character (!); the exclamation mark character and all following characters up to the end of the line are ignored. C language-style comments ("`/*...*/`") are also permitted and may span multiple lines.

Labels

A `label` is either a symbol or a single decimal digit *n* (0…9). A label is immediately followed by a *colon* (:).

Numeric labels may be defined repeatedly in an assembly file; normal symbolic labels may be defined only once.

A numeric label *n* is referenced after its definition (backward reference) as *n*`b`, and before its definition (forward reference) as *n*`f`.

Numbers

Decimal, hexadecimal, and octal numeric constants are recognized and are written as in the C language. However, integer suffixes (such as `L`) are not recognized.

For floating-point pseudo-operations, floating-point constants are written with `0r` or `0R` (where `r` or `R` means `REAL`) followed by a string acceptable to `atof(3)`; that is, an optional sign followed by a non-empty string of digits with optional decimal point and optional exponent.

The special names `0rnan` and `0rinf` represent the special floating-point values *Not-A-Number* (NaN) and *INFinity. Negative Not-A-Number* and *Negative INFinity* are specified as `0r-nan` and `0r-inf`.

Note – The names of these floating-point constants begin with the digit zero, *not* the letter "O."

2

Strings

A `string` is a sequence of characters quoted with either double-quote mark (`"`) or single-quote mark (`'`) characters. The sequence must not include a *newline* character. When used in an expression, the numeric value of a string is the numeric value of the ASCII representation of its first character.

The suggested style is to use *single quote mark* characters for the ASCII value of a single character, and *double quote mark* characters for quoted-string operands such as used by pseudo-ops. An example of assembly code in the suggested style is:

```
add %g1,'a'-'A',%g1 ! g1 + ('a' - 'A') --> g1
```

The escape codes described in Table 2-1, derived from ANSI C, are recognized in strings.

Table 2-1 Escape Codes Recognized in Strings

Escape Code	Description
\a	Alert
\b	Backspace
\f	Form feed
\n	Newline (line feed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
nnn	Octal value *nnn*
\x*nn...*	Hexadecimal value *nn...*

Symbol Names

The syntax for a symbol *name* is:

```
{ letter | _ | $ | . }   { letter | _ | $ | . | digit }*
```

In the above syntax:

- Uppercase and lowercase letters are distinct; the underscore (_), dollar sign ($), and dot (.) are treated as alphabetic characters.
- Symbol names that begin with a dot (.) are assumed to be local symbols. To simplify debugging, avoid using this type of symbol name in hand-coded assembly language routines.
- The symbol dot (.) is predefined and always refers to the address of the beginning of the current assembly language statement.
- External variable names beginning with the underscore character are reserved by the ANSI C Standard. Do *not* begin these names with the underscore; otherwise, the program will not conform to ANSI C and unpredictable behavior may result.

Special Symbols - Registers

Special symbol names begin with a *percentage sign* (`%`) to avoid conflict with user symbols. Table 2-2 lists these special symbol names.

Table 2-2 Special Symbol Names

Symbol Object	Name	Comment
General-purpose registers	`%r0 … %r31`	
General-purpose global registers	`%g0 … %g7`	Same as `%r0 … %r7`
General-purpose out registers	`%o0 … %o7`	Same as `%r8 … %r15`
General-purpose local registers	`%l0 … %l7`	Same as `%r16 … %r23`
General-purpose in registers	`%i0 … %i7`	Same as `%r24 … %r31`
Stack-pointer register	`%sp`	(`%sp = %o6 = %r14`)
Frame-pointer register	`%fp`	(`%fp = %i6 = %r30`)
Floating-point registers	`%f0 … %f31`	
Floating-point status register	`%fsr`	
Front of floating-point queue	`%fq`	
Coprocessor registers	`%c0 … %c31`	
Coprocessor status register	`%csr`	
Coprocessor queue	`%cq`	
Program status register	`%psr`	
Trap vector base address register	`%tbr`	
Window invalid mask	`%wim`	
Y register	`%y`	

Table 2-2 Special Symbol Names *(Continued)*

Symbol Object	Name	Comment
Unary operators	`%lo`	Extracts least significant 10 bits
	`%hi`	Extracts most significant 22 bits
	`%r_disp32`	Used only in Sun compiler-generated code.
	`%r_plt32`	Used only in Sun compiler-generated code.
Ancillary state registers	`%asr1` … `%asr31`	

There is no case distinction in special symbols; for example,

```
%PSR
```

is equivalent to

```
%psr
```

The suggested style is to use lowercase letters.

The lack of case distinction allows for the use of non-recursive preprocessor substitutions, for example:

```
#define psr %PSR
```

The special symbols `%hi` and `%lo` are true unary operators which can be used in any expression and, as other unary operators, have higher precedence than binary operations. For example:

```
%hi a+b  =  (%hi a)+b
%lo a+b  =  (%lo a)+b
```

To avoid ambiguity, enclose operands of the `%hi` or `%lo` operators in parentheses. For example:

```
%hi(a) + b
```

Operators and Expressions

The operators described in Table 2-3 are recognized in constant expressions.

Table 2-3 Operators Recognized in Constant Expressions

Binary	**Operators**	**Unary**	**Operators**
+	Integer addition	+	(No effect)
–	Integer subtraction	–	2's Complement
*	Integer multiplication	~	1's Complement
/	Integer division	`%lo`	Extract least significant 10 bits
%	Modulo	`%hi`	Extract most significant 22 bits
^	Exclusive `OR`	`%r_disp32`	Used in Sun compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.
<<	Left shift	`%r_plt32`	Used in Sun compiler-generated code only to instruct the assembler to generate specific relocation information for the given expression.
>>	Right shift		

Table 2-3 Operators Recognized in Constant Expressions *(Continued)*

Binary	Operators	Unary	Operators
&	Bitwise `AND`		
|	Bitwise `OR`		

Since these operators have the same precedence as in the C language, put expressions in parentheses to avoid ambiguity.

To avoid confusion with register names or with the `%hi`, `%lo`, `%r_disp32`, or `%r_plt32` operators, the modulo operator `%` must *not* be immediately followed by a letter or digit. The modulo operator is typically followed by a space or left parenthesis character.

Assembler Error Messages

Messages generated by the assembler are generally self-explanatory and give sufficient information to allow correction of a problem.

Certain conditions will cause the assembler to issue warnings associated with delay slots following Control Transfer Instructions (CTI). These warnings are:

- Set synthetic instructions in delay slots
- Labels in delay slots
- Segments that end in control transfer instructions

These warnings point to places where a problem could exist. If you have intentionally written code this way, you can insert an `.empty` pseudo-operation immediately after the control transfer instruction.

The `.empty` pseudo-operation in a delay slot tells the assembler that the delay slot can be empty or can contain whatever follows because you have verified that either the code is correct or the content of the delay slot does not matter.